MA 3046 - Matrix Analysis
Laboratory Number 6
Condition, Stability and "Good" and "Bad" Problems and Algorithms

As we have often noted before, numerical linear algebra involves considerations somewhat different from both those which arise in the introductory linearly courses and those which arise in the study of other numerical methods, such as the numerical approximation of integrals. In the case of the system of linear equations, i.e.,

$$\mathbf{A}\,\mathbf{x} \;=\; \mathbf{b} \tag{1}$$

a primary reason for these differing considerations is the fact that, according to introductory linear algebra theory, this problem is exactly solvable, by Gaussian elimination, in a finite number of steps. (For this reason, we call Gaussian Elimination a *direct* method.) However, as we have discussed extensively discussed, "real world" applications tend to involve very large matrices, for which the resulting number of computations (and hence the time) required to perform a given algorithm can become truly daunting, even on a "fast" computer. Moreover, as we have also discussed, computers' floating-point number systems are inexact, and while the individual errors inherent in any single operation may be negligible, the accumulated effects of such errors may become severe when a large number of computations must be performed. Lastly, the actual data, e.g. in this case the values of $\mathbf{A}$ and $\mathbf{b}$ which we "feed" to a particular computer for a given problem are almost certain inexact, due to such causes as measurement errors, production quality variations, variations allowed under manufacturer's specifications, etc. All of these latter effects, which are generally many orders of magnitude larger than machine precision, introduce further inaccuracies into the solution process.

As we have already discussed in class, the degree to which the relative changes in the data of a problem are reflected in relative changes in the solution of that problem are referred to as the *condition* of that problem, and are expressed, quantitatively, by the condition number of that problem. For (1) is given by

$$\kappa(\mathbf{A}) = \| \mathbf{A} \| \| \mathbf{A}^{-1} \| \tag{2}$$

and relative changes (or errors) in the data can be *amplified* by an factor up to the condition number. Therefore, colloquially, we shall consider an algorithm "good" if it has a "small" condition number (relative to the order of magnitude of the errors we expect), and "bad" if it does not. Mathematically, we refer to these cases as well-conditioned, and ill-conditioned, respectively.

The complementary consideration is how well can we expect an actual algorithm, i.e. some specified sequence of computations, implemented in a computer program, to perform, given the inaccuracies of floating point arithmetic, assuming the intial data are correct. In general, this question is addressed in terms of discussing the *stability* of the algorithm. We

call an algorithm *forward stable* if it can (relatively) accurately solve any problem, i.e., in the case of (1), if we can *always* expect the computed solution, $\tilde{\mathbf{x}}$, to satisfy

$$\frac{\| \mathbf{x} - \tilde{\mathbf{x}} \|}{\| \mathbf{x} \|} \leq C \epsilon_{\text{machine}} \tag{3}$$

and *backward stable* if the computed solution *always* exactly solves a problem with data close to the data used by the algorithm, i.e. again in the case of (1), if

$$\tilde{\mathbf{A}}\tilde{\mathbf{x}} = \tilde{\mathbf{b}} \quad \text{where} \quad \left[\frac{\| \mathbf{A} - \tilde{\mathbf{A}} \|}{\| \mathbf{A} \|} + \frac{\| \mathbf{b} - \tilde{\mathbf{b}} \|}{\| \mathbf{b} \|} \right] \leq C' \epsilon_{\text{machine}} \tag{4}$$

where C and C' are constants, and $\mathbf{x}$ denotes the exact (infinite precision) solution for the given data. (For an algorithm to be practically backward stable, generally we need C' to be relatively small, considering machine precision, say, for MATLAB, on the order of magnitude of tens to thousands.) Analysis shows that forward stability is virtually impossible to achieve, since a truly ill-conditioned problem can render virtually any algorithm useless. Backward stability, however, is not only often relatively easy to prove, but it also relatively easy to show that, for a backward stable algorithm

$$\frac{\| \mathbf{x} - \tilde{\mathbf{x}} \|}{\| \mathbf{x} \|} \leq C' \kappa(A) \epsilon_{\text{machine}} \tag{5}$$

or, colloquially

> *A practically backward stable algorithm will always produce a reasonably accurate solution to a reasonably well-conditioned problem*

In this laboratory, we shall investigate both simulated low-precision machines and full MATLAB precision.

Basic Gaussian elimination, of course, using *elementary row operations*, i.e.

$$R_k \leftarrow R_k - l_{kj} R_i \tag{6}$$

to reduce an augmented matrix to (augmented) echelon form. The l_{kj} multipliers here do depend on the values of the elements in the column being eliminated, and specifically the *denominator* of l_{kj} is the coefficient (a_{jj}) in the pivot position of that column. (Unless, of course that pivot element is zero, in which case some rows must be interchanged before proceeding.)

Floating-point arithmetic introduces several potentially serious problems because of fundamental aspects of Gaussian elimination:

- First, because elementary row operations require (potentially catastrophic cancellation) subtractions, elimination may fail to accurately compute zero pivots, producing instead only "small" ones.

- Secondly, because of this, critical decisions in Gaussian elimination, e.g. row interchanges, that depend on whether or not elements are **exactly** zero, may not correctly made.
- Finally, if one of the Gaussian elimination multipliers (l_{kj}) ever become sufficiently large, because of a small pivot, even though the pivot is not theoretically exactly zero, then, numerically, the elementary row operation (6) may effectively produce

  $$R_k \leftarrow (-l_{kj})\, R_i$$

  which results in the effective loss of most or even all of the "information" in R_k, and, moreover, produces a (near) singular matrix!

In this laboratory, we shall use the previously-introduced variants of Gaussian elimination contained in programs **ge_basic.m**, which performs step-by-step elimination in full MATLAB precision, and **ge_steps_chop.m**, which uses **chop( )** to simulate Gaussian elimination with row interchanges only to avoid zero pivots in finite-precision machine, to further study some of these specific aspects. (Because those programs were introduced in previous labs, we don't list them here again.)

We must reemphasize, however, that, in general, there is seldom a good reason to write a special computer program to solve a problem for which good, commercial-grade codes already exist. Gaussian elimination is an excellent example of this. MATLAB's built-in backslash (\) command provides an extremely robust, high-quality solver for

$$\mathbf{A}\,\mathbf{x} \;=\; \mathbf{b}$$

and we strongly encourage its use for virtually any instance of system (1). The only reason we use such "home-grown" Gaussian elimination codes as **ge_steps_chop.m** is because MATLAB is simply too accurate ($\epsilon_{\text{machine}} \doteq 10^{-16}$) to allow us to easily observe the effects of floating-point errors, and how these effects can propagate through an algorithm.

Therefore, in the second part of the laboratory, we will consider the effect of condition on MATLAB's standard backslash operator, which, as we have noted before, utilizes a variant of Gaussian elimination to solve (1) when **A** is square. We do this in next script, **cond_nbr.m** (Figure 6.4), which uses the backslash (\), **norm()** and **cond()** commands, as well as **rand()**, to investigate the fundamental accuracy inequality (5) above. It does this by generating two hundred random fifty by fifty matrices (**A**). For each, a right-hand vector (**b**) corresponding to the solution

$$\mathbf{xtrue} = \;[\;\mathbf{1}\;\mathbf{1}\;\mathbf{1}\;\ldots\;\mathbf{1}\;]^{\mathbf{T}}$$

is generated by "normal" matrix multiplication, i.e.

$$\mathbf{b} = \mathbf{A} \cdot \mathbf{xtrue}$$

(Note that this calculation will be simply be equivalent to summing each row of **A**, and because **rand( )** ensures all of the elements of **A** are positive, this sum should be accurate

```
   NA   = 50 ;
   data = [ ] ;
%
   for n = 1:200 ;
      a = rand(NA) ;
%
      xtrue = ones(NA,1);
      b     = a*xtrue ;
%
      xcalc = a\b ;
      reler = norm(xtrue-xcalc)/norm(xtrue) ;
%
      data = [ data ; cond(a)  reler ] ;
%
   end
%
   loglog( data(:,1) , data(:,2) , '*' )
   xlabel('cond(a)') ;
   ylabel('norm(e)/norm(x)') ;
   title('Relation of Relative Error to Condition Number')
```

Figure 6.4 - Listing of Program **cond_nbr.m**

to full machine precision.) With this computed right-hand side, the resulting system of the form (1) is then "solved," using MATLAB's backslash command, for a computed solution, **xcalc**. (This solution will, in general, not be exactly correct due to the accumulated effects of round-off errors in the Gaussian elimination process.) The relative accuracies of each of these solutions is then computed and graphically displayed relative to the condition number of the associated matrix. (Note that both **norm()** and **cond()** in MATLAB implicitly use the Euclidean (i.e., $\|\cdot\|_2$) norm, rather than the infinity or row-sum norm (i.e., $\|\cdot\|_\infty$) that we also commonly use. While these two norms are not the same, they produce sufficiently similar results that we can use either here.)

Name: _____

MA 3046 - Matrix Analysis
Laboratory Number 6
Condition, Stability and "Good" and "Bad" Problems and Algorithms

1. Copy to your local directory the file:

cond_nbr.m

Be sure that you still have, on your disk for earlier labs, the programs

ge_basic.m , **ge_steps_chop.m** and **bwd_solve_chop.m**

Then start MATLAB.

2. Using your texteditor to open the **ge_basic.m** and **ge_steps_chop.m** script and review them until you are fairly comfortable with the flow of MATLAB logic and the computations.

3. Give the MATLAB commands

clear
global NDIGITS
NDIGITS = 3

Then create the 4×4 matrix

$$\mathbf{A} = \begin{bmatrix} 2.03 & 1.00 & 9.32 & 5.25 \\ -2.01 & -0.98 & -10.5 & -3.31 \\ 6.04 & 7.47 & 4.19 & 6.72 \\ 2.72 & 4.45 & 8.46 & 8.38 \end{bmatrix}$$

and the column vector:

$$\mathbf{b} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

4. For the matrix determined in part 3, determine the condition number

$$\kappa(\mathbf{A})$$

5. For the matrix and vector created in part 3, use the MATLAB backslash function to solve $\mathbf{A}\,\mathbf{x} = \mathbf{b}$, and record the result

$$\mathbf{x}_{true} = \begin{bmatrix} \\ \\ \\ \\ \\ \end{bmatrix}$$

6. Using the vector $\mathbf{x}_{true}$ computed in part 5, and the original right-hand side, compute:

$$\frac{\| \mathbf{b} - \mathbf{A}\mathbf{x}_{true} \|}{\| \mathbf{b} \|}$$

Briefly explain what this calculation means, and whether or not the result seems reasonable.

7. Now, again using the matrix and vector from part 3, run the MATLAB program **ge_basic.m** on the augmented matrix [**A** | **b**]. Look carefully for the appearance of small pivots, and the resulting effects.

Record the final echelon augmented matrix:

$$\mathbf{uwork} = \begin{bmatrix} & & & \\ & & & \\ & & & \\ & & & \end{bmatrix}$$

Based on this result, use the MATLAB backslash command, along with the proper submatrices of **uwork** to compute the solution resulting from using Gaussian elimination, without partial pivoting,in full MATLAB precision:

$$\mathbf{x}_{full} = \begin{bmatrix} \\ \\ \\ \\ \end{bmatrix}$$

Briefly describe how well this agrees with the answer computed in part 5, and why that should or should not have been expected.

8. Now run the MATLAB program **ge_steps_chop.m** on the same augmented matrix as in problem 7. Look now at what happens when a small pivots appears, and the resulting effects.

Based on the final echelon matrix:

$$\mathbf{uwork} = \begin{bmatrix} & & & \\ & & & \\ & & & \\ & & & \\ & & & \end{bmatrix}$$

compute the solution resulting from using Gaussian elimination, with partial pivoting, in a three-digit, decimal, rounding machine:

$$\tilde{\mathbf{x}}_4 = \begin{bmatrix} \\ \\ \\ \\ \end{bmatrix}$$

Compare the accuracy of this solution with that of the solutions obtained in parts 5-7 above.

9. Determine the condition number of the upper triangular Gaussian Elimination matrix **U** which is located in the left portion of the augmented matrix **uwork** computed in part 8.

$$\kappa(\mathbf{U})$$

Compare this to the value of the condition number of the original matrix **A**. What does this imply about possible behavior of "plain" Gaussian elimination?

10. Study the MATLAB script file **cond_nbr.m** to make sure you understand what it is doing. Then run it (this may take a little time) and observe the results. Does the observed behavior look reasonable and agree with the theory?